Quad-kd Trees: A General Framework for kd Trees and Quad Trees

Nikolett Bereczky[a], Amalia Duch[b,1,*], Krisztián Németh[a], Salvador Roura[b,1]

[a] *Budapest University of Technology and Economics*
Department of Telecommunications and Media Informatics
1117 Budapest, Magyar tudósok körútja 2.
Hungary
[b] *Technical University of Catalonia, Barcelona Tech*
Department of Computer Science
Campus Diagonal Nord, Building Omega
C. Jordi Girona 1-3, 08034 Barcelona
Catalonia, Spain

Abstract

We introduce the quad-kd tree: a general purpose and hierarchical data structure for the storage of multidimensional points. Quad-kd trees include point quad trees and kd trees as particular cases and therefore they could constitute a general framework for the study of fundamental properties of trees similar to them. Besides, quad-kd trees can be tuned by means of insertion heuristics and bucketing techniques to obtain trade-offs between their costs in time and space. We propose three such heuristics and we show analytically and experimentally their competitive performance. Our analytical results back the experimental outcomes and suggest that the quad-kd tree is a flexible data structure that can be tailored to the resource requirements of a given application.

Keywords: quad trees, kd trees, multidimensional data structures, associative retrieval.

1. Introduction

Let us consider a collection $\mathcal{F}$ of n records, where each *record* is an ordered k-tuple $x = (x_0, \ldots, x_{k-1})$ of values (the attributes or coordinates of the record) drawn from $D = \prod_{0 \le j < k} D_j$, where each D_j is a totally ordered domain.

*Corresponding author.
Email addresses: nikolett.bereczky@tmit.bme.hu (Nikolett Bereczky), duch@cs.upc.edu (Amalia Duch), krisztian.nemeth@tmit.bme.hu (Krisztián Németh), roura@cs.upc.edu (Salvador Roura)

[1] This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R) and AGAUR grant SGR 2014:1034 (ALBCOM).

	kd trees	quad trees
Space	$\Theta(n)$	$\Theta(2^k n)$
Time	$2n \ln n + O(n)$	$\frac{2}{k} n \ln n + O(n)$

Table 1: Space and Time of kd trees and quad trees of n nodes

A *query* over $\mathcal{F}$ is a retrieval of all records whose attributes satisfy certain conditions. The query is considered *associative* when it deals with more than one of the attributes. Examples of associative queries are: (i) *nearest-neighbor* queries, to retrieve the record in $\mathcal{F}$ closest to a given record under a given distance metric, (ii) *partial match* queries, to retrieve all records in $\mathcal{F}$ that match the attributes of the query that are specified, or (iii) *orthogonal range* queries, to retrieve all records in $\mathcal{F}$ that fall inside a given hyper-rectangle whose sides are parallel to the coordinate axes.

Associative retrieval is an important computing task [21, 34]. When several types of associative queries are required, general purpose multidimensional data structures, such as kd trees and quad trees[2], are typically used [21, 34]. Here, we are interested in the amount of memory required by the data structure (its *cost in space*), and in the execution time of each operation (its *cost in time*). Both measures strongly depend on the computer model under consideration and on the way of storing the data structures. We will follow Samet's [34] customary representation of multidimensional trees in main memory. Other approaches such as succinct representation of trees or the consideration of memory hierarchies are beyond the scope of this paper. Nevertheless, the techniques used in those research areas are applicable to quad-kd trees—the data structure introduced in this paper—in the same way that apply to kd trees and quad trees (see Chapters 27, 34 and 37 of [27] and [2, 9, 11, 31]).

As shown in Table 1, both kd trees and quad trees require a basic amount of $\Theta(kn)$ memory to store n k-dimensional records. In addition, kd trees are binary trees, while quad trees are 2^k-ary trees, so the space overhead due to pointers is $\Theta(n)$ and $\Theta(2^k n)$, respectively. Note that, for large k, the former is diminishable, while the latter is prohibitive, mainly because most pointers just point to empty subtrees [25].

Regarding the execution time, there are several results in the literature analyzing the cost of performing insertions, deletions, exact match, partial match, orthogonal or nearest neighbor queries in both kd trees (and a wide range of their variants) and quad trees [7, 8, 10, 12, 13, 14, 16, 19, 26]. A measure closely related to the time performance in almost any operation is the average height of the tree, or equivalently, its Internal Path Length (*IPL* from now on) [23]. It turns out that the *IPL* of perfectly balanced or random quad trees is asymptotically optimal, while that of perfectly balanced or random kd trees is considerably

[2]Throughout this work, we will write "quad trees" to refer to point quad trees [34].

larger [24](see Table 1). It is worth-while to mention that the leading terms of $n \ln n$ shown in Table 1 only matter when n is very large, because otherwise the implied constants in the O-term might play a more significant role (for example, they may be larger than $\ln n$).

From the above discussion, and very roughly speaking, one could argue that kd trees are space-optimal while quad trees are *IPL*-optimal. It is a natural question, therefore, whether it is possible to characterize the trade-off between the *IPL* and the number of empty subtrees of these multidimensional trees.

Regarding other previous work, there are some ad-hoc multidimensional data structures for specific associative queries [4, 21, 22, 32, 37]. However, some of these techniques substantially increment the cost in space, and it is often the case that ad-hoc data structures do not perform well in a general setting.

There are also several proposals to keep general multidimensional trees balanced in order to optimize either the execution time of some queries or the required space [10, 12, 17, 18, 21, 28, 29, 38, 36]. Unfortunately, in such settings the cost of update operations increases significantly.

Other works adapt the behavior of general purpose multidimensional data structures either to the distribution of the input data or to the most frequent kind of multidimensional queries that they have to support [15, 21, 30], at the cost of increments in space and in the execution time of the update operations.

To the best of our knowledge, there is no work combining simultaneously tunable space/time compromise as well as adaptability to the distribution of the input data. Quad-kd trees are general purpose, flexible multidimensional trees that allow us to balance the space and time costs. Using insertion heuristics (we propose three of them) it is possible to produce the whole range of trade-offs that lie between kd trees and quad trees. Finally, by means of the bucketing technique, it is possible to obtain considerable savings in space.

The rest of the paper is organized as follows. We start with some preliminaries in Section 2. Then, in Section 3, we formally introduce quad-kd trees together with their implementation. Afterwards, in Section 4, we present the *Random-Split*, the *Distance-Dependent* and the *Probability-Dependent* insertion heuristics as well as their experimental performance. In Section 5 we provide the theoretical value of the *IPL* of random quad-kd trees built using the proposed heuristics, while in Section 6 we study the number of empty subtrees and propose a simple bucketing heuristic to considerably reduce the amount of wasted space. Section 7 shows the experimental performance of quad-kd trees built with real data. Our final remarks and proposals of further work can be found in Section 8. A preliminary version of this work was published in [6].

2. Quad trees and kd trees

We devote this section to summarize some definitions and properties that will be useful for the forthcoming sections. Assume, w.l.o.g., that no two keys have the same coordinates in any of the dimensions. For convenience, sometimes we will also assume that the search domain is $D = [0, 1]^k$.

Definition 1 (Bentley [3]). A k-dimensional tree (or kd tree) T of size $n \geq 0$ is a binary tree such that

- it is either empty when $n = 0$, or
- its root stores a record with key x and has a discriminant j, $0 \leq j < k$, and the remaining $n - 1$ records are stored in the left and right subtrees of T, say L and R, in such a way that: both L and R are kd trees, for any key $u \in L$, it holds that $u_j < x_j$ and, for any key $v \in R$, it holds that $x_j < v_j$.

Let $\langle x, j \rangle$ denote a node that contains a key x with discriminant j. A kd tree of size n induces a partition of the domain D into $n + 1$ regions, each corresponding to a leaf (or equivalently empty subtree throughout this work) in the kd tree. The *bounding box* (or *bounds array*) of $\langle x, j \rangle$ is the region of the space delimited by the leaf in which x falls when it is inserted into the tree. Thus, if the root is $\langle y, i \rangle$, its bounding box is $[0, 1]^k$, the bounding box of its left subtree is $[0, 1] \times \cdots \times [0, y_i] \times \cdots \times [0, 1]$, and so on.

Different variants of kd trees have been proposed so far; many differ in the way in which discriminants are assigned to nodes. In the original or *standard* kd trees by Bentley [3], the root of the tree gets discriminant 0, those in the first level get 1, ..., those in the $(k - 1)$-th level get $k - 1$, those in the k-th level get 0, and so on, in this cyclic way. Note that, because of this fixed rule, there is no need to explicitly store the discriminants. Duch et al. [13] proposed *relaxed* kd trees, where each node is assigned a random discriminant, uniform and independently drawn from $\{0, \ldots, k-1\}$. The *squarish* kd trees of Devroye et al. [12] try to get a more balanced partition of the space by discriminating along the coordinate for which the bounding box of the node is more elongated.

Definition 2 (Bentley and Finkel [5]). A quad tree T of size $n \geq 0$ is a 2^k-ary tree such that

- it is either empty when $n = 0$, or
- its root stores a record with key x and has 2^k subtrees, each one associated to a k-bitstring $w = w_0 w_1 \ldots w_{k-1} \in \{0, 1\}^k$, and the remaining $n - 1$ records are stored in one of these subtrees, let's say T_w, in such a way that $\forall w \in \{0, 1\}^k$: T_w is a quad tree, and for any key $y \in T_w$, it holds that $y_j < x_j$ if $w_j = 0$ and $y_j > x_j$ otherwise, for all $0 \leq j < k$.

A quad tree of size n induces a partition of D into $(2^k - 1)n + 1$ regions, each corresponding to a leaf in the quad tree. The *bounding box* of a node in a quad tree is defined in a similar way as for kd trees.

Because of their definition, the insertion and search algorithms for kd trees and quad trees are straightforward. For insertions, the only difference between each variant of kd trees is the way in which discriminants are assigned to newly inserted nodes.

Regarding the probabilistic model generally used to analyze the expected performance of kd trees and quad trees, we have the following definition.

Definition 3. A kd tree or a quad tree with n keys is *random* if it is built with identical probability from any of the $n!^k$ possible input sequences.

As a consequence, the insertion of n points, in which every coordinate is independently drawn from a continuous distribution in $[0,1]^k$, into an initially empty tree produces random kd trees or random quad trees.

3. Quad-kd trees

Observe that the nodes of kd trees discriminate by exactly one coordinate, while the nodes of quad trees discriminate by all their k coordinates. Thus, it is natural to envisage a generalization, where the number of discriminating coordinates for each node can be any value between 1 and k.

Definition 4. A Quad-kd tree T of size $n \geq 0$ is such that

- it is either empty when $n = 0$, or
- its root stores both a record with key x and a coordinate split boolean vector $s = (s_0, \ldots, s_{k-1})$ that contains i ones, where $1 \leq i \leq k$ is the order of s, and the node has 2^i subtrees that store the $n-1$ remaining records as follows: each subtree T_w is associated to a string $w = w_0 \ldots w_{k-1}$ with $w \in \{0, 1, \#\}^k$ such that T_w is a quad-kd tree and, for any key $y \in T_w$ and $0 \leq j < k$, it holds that
  - $w_j = \#$ iff $s_j = 0$
  - if $s_j = 1$ and $w_j = 0$, then $y_j < x_j$
  - if $s_j = 1$ and $w_j = 1$, then $y_j > x_j$.

Figure 1 includes an example of a 3-dimensional quad-kd tree. Each node has its key between brackets and its split boolean vector in parenthesis, and every edge shows the string w of the subtree it points to.

The algorithms for exact search and insertion are similar to those of kd trees and quad trees. To search for a record in a quad-kd tree, we start at the root, and examine the values in the split vector: for every 1 we compare the corresponding coordinate of the root with the one of the record we are looking for. Afterwards, we recursively search in the subtree whose associated string matches the result of all the comparisons, until we find the requested record (successful search) or an empty subtree (unsuccessful search).

To insert a new record, we search for it as described above, until we reach an empty subtree, which we replace by a new node with the key to be inserted and empty children. At this point, a predefined algorithm to generate the new split vector is required. In the next section we propose three such algorithms.

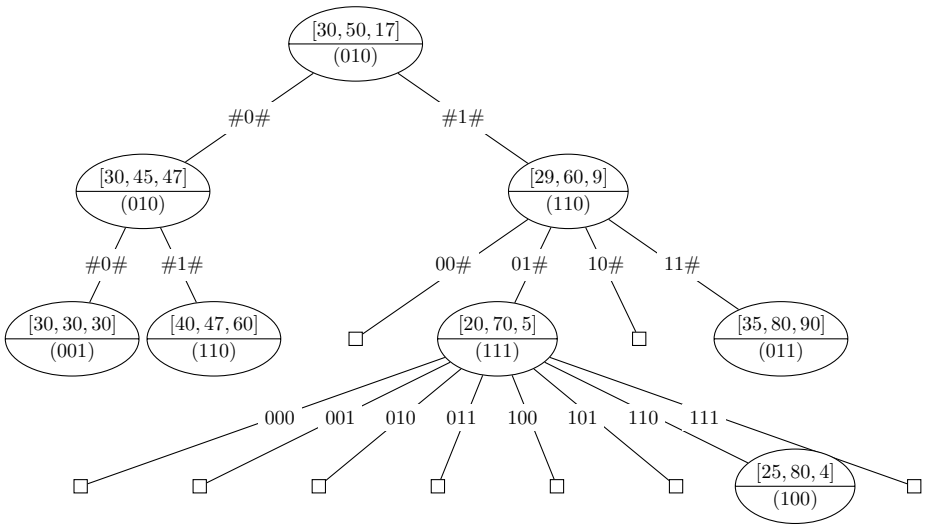

Figure 1: An example of 3-dimensional quad-kd tree (with some leaves omitted to simplify the figure)

4. Insertion heuristics

Here we introduce the *Random-Split* (*RS*), *Distance-Dependent* (*DD*) and *Probability-Dependent* (*PD*) insertion heuristics. As we will show, any of these heuristics provides a "smooth" transition between the space and the *IPL* of kd trees and of quad trees, so quad-kd trees can be tuned for specific application requirements.

Random-Split. *RS* generates the bits of each split vector randomly and independently, using the same Bernoulli distribution for each coordinate, where the probability of occurrence of the value 0 is a given constant referred to as *Prob-of-0* or simply p. Note that it can happen that some split vector is generated with all zeros. In that case, a randomly chosen position is set to one. A simple implementation of this heuristic is included in Program 1.

In Figure 2 we show the performance of quad-kd trees built using *RS* and *DD*. Specifically, we show experimental results obtained by comparing the *IPL* (as a measure of data access time) and the number of empty subtrees (as a measure of storage space usage) of random kd trees and quad trees versus quad-kd trees built using our proposed heuristics. All measures in this section, unless otherwise stated, were obtained by averaging the values of 100 3-dimensional trees with $n = 10^5$. Experiments for 2, 4 and 5-dimensional trees, although not included, produced equivalent results.

We start by comparing quad-kd trees built with independent uniformly distributed keys from a continuous domain, with kd trees and quad trees built with the same set of keys inserted in the same order. As expected, we can see

Program 1 Implementation of the Random-Split insertion heuristic.

```
typedef struct node* tree;

struct node {
  vector<int> key;    // k-dimensional key
  vector<int> disc;   // vector of discriminants
  vector<tree> T;     // vector of subtrees
  int elems;          // number of elements of the tree
};

void insert(tree& t, const vector<int>& key) {
  if (not t) { // if the tree is empty
    t = new node;
    t->key = key;

    for (int i = 0; i < k; ++i)
      // function prob() decides for every coordinate
      // whether or not  it will discriminate
      if (prob()) t->dim.push_back(i);
    int m = t->disc.size();
    if (m == 0) {
      t->disc.push_back(rand()%k);
      m = 1;
    }

    t->T = vector<tree>(1<<m);
    t->elems = 1;
    return;
  }

  // computation of the corresponding subtree number
  int masc = 0;
  int m = t->disc.size();
  for (int j = 0; j < m; ++j) {
    int i = t->disc[j];
    if (key[i] > t->key[i]) masc |= (1<<j);
  }

  insert(t->T[masc], key);
  ++t->elems;
}
```

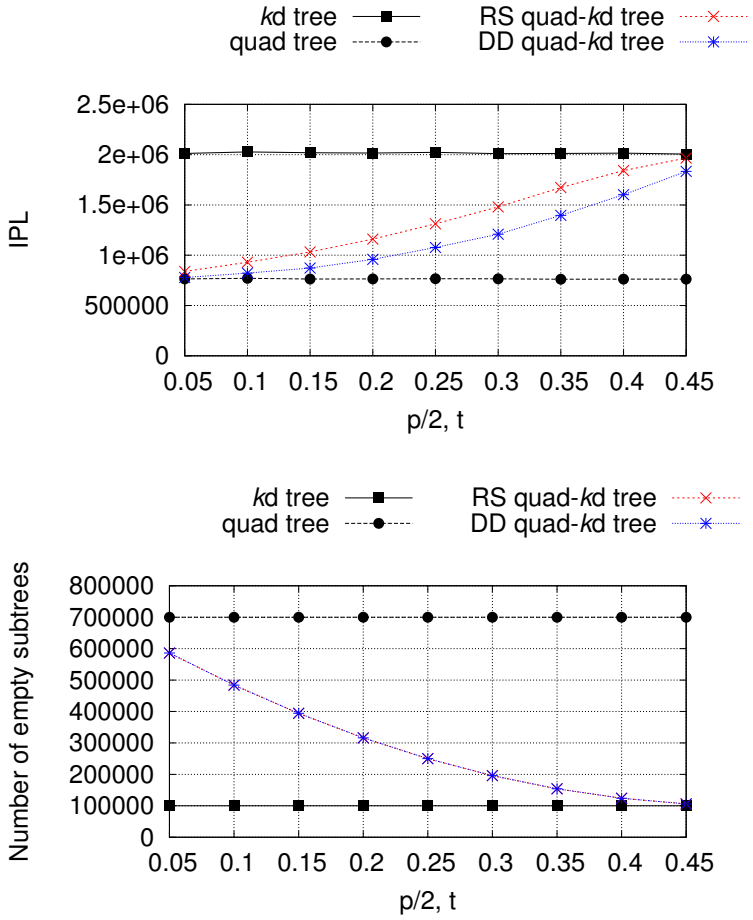

Figure 2: Performance of *RS* and *DD*

that as p increases the *IPL* of the quad-kd tree grows and the space it needs diminishes. Indeed, the more ones we have in the split vector, the more quad-kd trees resemble quad trees (case in which $p = 0$), and the less ones we have, the more quad-kd trees resemble kd trees (case in which $p = 1$).

Distance-Dependent. For a given node with key $x = (x_0, \ldots, x_{k-1})$ and bounding box $[\ell_0, u_0] \times \ldots \times [\ell_{k-1}, u_{k-1}]$, *DD* decides whether to discriminate or not by each coordinate $0 \leq j < k$ depending on the distance of x_j from both ℓ_j and u_j. More precisely, given a fixed value $0 \leq t \leq 0.5$ (the *Split-Threshold*), we discriminate by j if and only if $\frac{x_j - \ell_j}{u_j - \ell_j} \geq t$ and $\frac{u_j - x_j}{u_j - \ell_j} \geq t$. For example, let the root of a 2-dimensional tree have key $(0.35, 0.52)$ and bounds array $[0, 1] \times [0, 1]$. If the value of t is 0.4, then we only discriminate by the second coordinate—and consequently $s = (0, 1)$—because $0.35 < 0.40$ and $0.40 < 0.52 < 0.60$. Again, if

every coordinate in the split vector happens to be zero, a randomly chosen one is set to one.

Note that as the value of t grows the probability of splitting decreases. The idea behind this heuristic is to split only when the coordinate is useful enough to discriminate. For uniformly distributed keys, we prefer coordinates close to the center of the bounding box of its dimension, because the nodes inserted beneath will probably be evenly distributed in the forthcoming subtrees.

In the case of uniformly distributed keys, *DD* performs well. In Figure 2 we see that quad-kd trees are between quad trees and kd trees considering both the *IPL* and the number of the empty subtrees. When the value of t grows, the *IPL* increases and the space requirements diminish. It is worth mentioning that half of p and t are on the x axis of Figure 2 to allow comparison.

Comparing *RS* against *DD* for randomly built trees we can observe that the *IPL* of the latter is better. On the other hand, the number of empty subtrees is practically identical (the curves are overlapping and can't be distinguished in black and white print). These together mean that the more sophisticated *DD* outperforms *RS* in a random uniform setting. However, in applications for where the extra *IPL* is not a problem, it might be worth using *RS* because of its simplicity.

Probability-Dependent. *PD* is an extension of *DD* for keys drawn from distributions that do not produce random trees in the sense that the points (and their coordinates) are not independently generated (see Definition 3). When the assumption of independence holds *PD* and *DD* are equivalent. In this heuristic, given a node with key x, its j-th coordinate is used to discriminate only if both quotients $\frac{F(x_j)-F(\ell_j)}{F(u_j)-F(\ell_j)}$ and $\frac{F(u_j)-F(x_j)}{F(u_j)-F(\ell_j)}$ are greater than a given *Split-Threshold* constant, where F is the cumulative distribution function and u_j and ℓ_j are the coordinate bounds. Informally, the j-th coordinate of a given key is used to discriminate if it is somehow "centered" within the probability distribution of the keys.

We carried out experiments comparing kd trees and quad trees with quad-kd trees built using *RS*, *DD* and *PD*. We used keys generated from exponential and normal distributions. The latter is commonly used for modeling non-uniformly distributed data in multidimensional settings [1].

For the exponential distribution, in Figure 3 (top) we can observe the effect on the performance of the expected value of the distribution μ. The results show that for smaller values of μ –corresponding to higher non-uniformity– *DD* produces a larger *IPL* but less empty subtrees (approaching the behavior of kd trees), while these two values remain constant for *RS* and *PD*. It seems, therefore, that using *DD* is not suitable in highly non-uniform settings. By contrast, *RS* and *PD* seem to be very competitive since both of their performances appear to be independent of μ. Regarding the *IPL*, as expected, *PD* has a better performance than *RS*, but their performance is similar regarding the number of empty subtrees. Nevertheless, because of its simplicity, *RS* might be preferable in settings where its *IPL* overhead is acceptable.

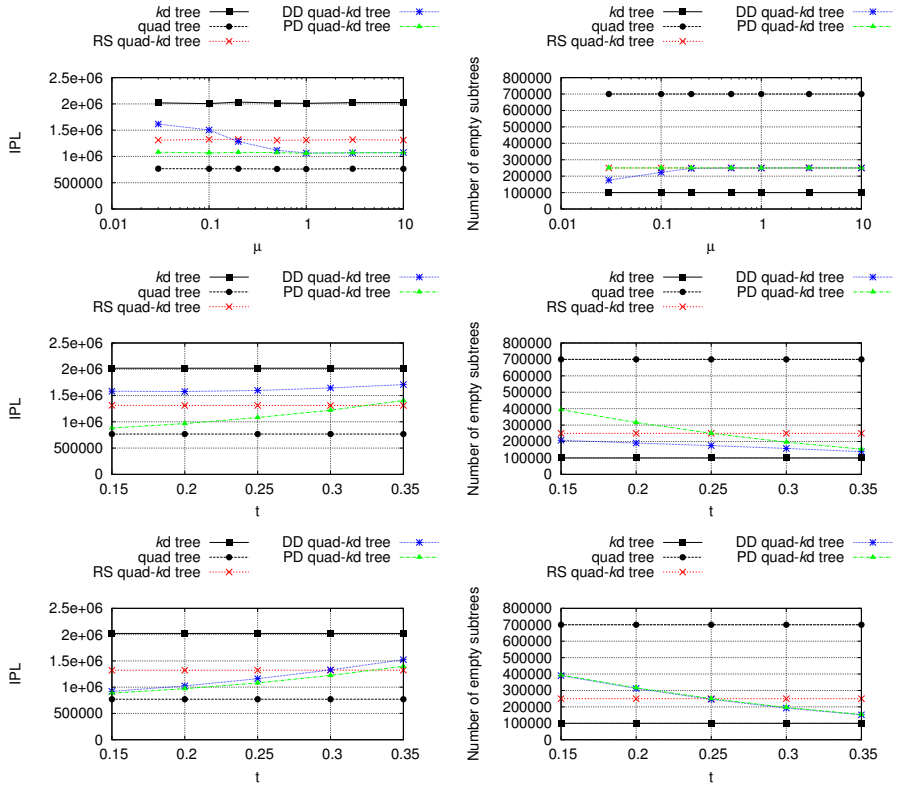

Figure 3: Performances of RS, DD and PD, $p = 0.5$. Exponential distribution: with variable μ (top), with $\mu = 0.03$ (middle). Gaussian distribution with $\sigma^2 = 0.003$, $\mu = 0.25$ (bottom)

Fixing μ (or σ and μ, for trees built from a Gaussian distribution) and changing the value of the *Split-Threshold*, the experimental results (see Figure 3, middle and bottom) show that, as expected, *PD* outperforms *DD* with respect to the *IPL*. On the other hand, the diagrams indicate that *DD* is better with respect to the number of empty subtrees.

5. *IPL*

Here we study the asymptotic average *IPL* of random quad-kd trees built under *RS* and *DD*, using the same probabilistic model generally used to analyze the expected performance of kd trees and quad trees (see Definition 3). Since the quad-kd trees produced using *PD* are not random (in the sense required by Definition 3), *PD* is not analyzable formally using classical and well established techniques, which all rely under this assumption. Moreover, *PD* is so general that its mathematical analysis –if possible– would depend on the specific probability distribution of the input, i.e., it would be ad-hoc to each probability distribution and therefore out of the scope of this work.

It is well known that the *IPL* of both random kd trees and random quad trees of n nodes is of the form $cn \ln n + c'n + o(n)$, where c and c' are constants with values $c = 2, c' = -2.8$ for kd trees [35], and $c = 2/k$ for k-dimensional quad trees, with $c' = -0.6$ for $k = 2$ [20]. In this section we observe that the *IPL* of random quad-kd trees has a similar form. Indeed, we prove that it is of the form $\sim cn \ln n$, and we give a close formula for the value of $c = c(p, k)$ for *RS* and $c = c(t, 2)$ for *DD*.

We are going to proceed by giving the expected cost of a random exact search in random kd trees and random quad trees, and extending the results to quad-kd trees. This will allow us to derive the *IPL*, since it is well known that the *IPL* of a random tree with n keys is just n times the expected cost C_n of a successful search. Our asymptotic analysis relies in Roura's Continuous Master Theorem [33] and similar techniques. More sophisticated methods for solving recurrences (as for instance analytic combinatorics) could provide more accurate results, but they seem difficult to apply to quad-kd trees.

Let us start with a recurrence for the expected cost C_n of a random exact search in a random binary search tree (or in a random kd tree) of size n, where i designates the number of keys in the left subtree:

$$C_n = 1 + \sum_{i=0}^{n-1} \frac{1}{n} \left(\frac{i}{n} C_i + \frac{n-i-1}{n} C_{n-i-1} \right) = 1 + \sum_{i=0}^{n-1} \frac{2i}{n^2} C_i \ . \qquad (1)$$

If we denote by ω_i the weight of C_i, i.e., $\omega_i = 2i/n^2$, we observe that those weights asymptotically fit the shape of the function $f_1(z) = 2z$ between [0..1], (see top left of Figure 4), in the sense that $\omega_i = f_1(i/n)/n \simeq \int_{i/n}^{(i+1)/n} f_1(z)dz$. Informally speaking, $f_1(z)$ is like a continuous probability distribution, with indeed $\int_0^1 f_1(z)dz = 1$.

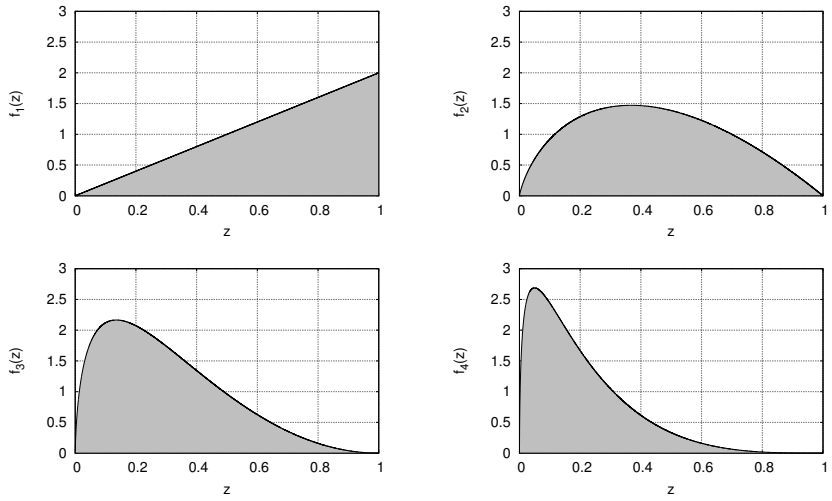

Figure 4: $f_1(z)$ (top left), $f_2(z)$ (top right), $f_3(z)$ (bottom left) and $f_4(z)$ (bottom right)

As shown in [33], a recurrence like (1) has always a solution of the form $C_n \sim c \ln n$ for some constant c. Moreover, the results in [33] also imply that solving the approximate equation

$$\begin{aligned} c \ln n &\simeq 1 + \sum_{i=0}^{n-1} \omega_i c \ln i \simeq 1 + \int_0^1 f_1(z) c \ln(zn) dz \\ &= 1 + c \int_0^1 f_1(z) \ln z dz + c \ln n \int_0^1 f_1(z) dz \end{aligned}$$

in fact provides the exact value for c. Since $\int_0^1 f_1(z)dz = 1$, the terms $c \ln n$ vanish and we get

$$0 = 1 + c \left[z^2 \ln z - \frac{z^2}{2} \right]_0^1 = 1 + c \left(-\frac{1}{2} \right) \ ,$$

which implies $c = 2$. The expected cost is certainly $\sim 2 \ln n = 2 \ln 2 \log_2 n \simeq 1.39 \log_2 n$. Therefore, the next theorem follows.

Theorem 1. *The expected* IPL *of a random kd tree of size n is $\sim 2n \ln n$.*

Let $f_k(z)$ be the function that describes the asymptotic shape of the weights of the recurrence for the average cost to search in a quad tree with k dimensions. We already have $f_1(z) = 2z$. To compute $f_2(z)$, note that using two dimensions to discriminate is like using each of the two dimensions one after another. The "density of probability" to reach some z from 1 in two steps is therefore $f_2(z) = \int_z^1 f_1(x) \frac{f_1(z/x)}{x} dx$. (For every x between z and 1, it is the "probability" to reach x in one step—$f_1(x)$—times the "probability" to reach z/x in another step,

which is $f_1(z/x)$ scaled by the factor $1/x$ so that the integral of the probability distribution f_1 between 0 and x adds up to 1.) Hence, $f_2(z) = 4z\int_z^1 1/x dx = -4z\ln z$. At the top right of Figure 4 we can see $f_2(z)$. Note how the weights are closer to the left than in $f_1(z)$, while the area $\int_0^1 f_2(z)dz$ is also 1.

For comparison with the traditional (discrete) approach, the recurrence for the cost of a search in a two-dimensional quad-tree can be seen to be exactly (see [20], page 102)

$$C_n = 1 + \sum_{i=0}^{n-1} \frac{4i}{n^2}(H_n - H_i)C_i \ ,$$

where $H_n \sim \ln n$ denotes as usual the n-th harmonic number $\sum_{i=1}^{n} 1/i$. Note how the weights fit the function $f_2(z)$:

$$4\frac{i}{n}\frac{H_n - H_i}{n} \sim -4\frac{i}{n}\frac{\ln(i/n)}{n} = \frac{f_2(i/n)}{n} \ .$$

Lemma 1. *For $k \geq 1$ and $K_k = -(-2)^k/(k-1)!$, $f_k(z) = K_k z(\ln z)^{k-1}$.*

PROOF. The proof is by induction. We already know that $f_1(z) = K_1 z(\ln z)^0$. Assuming that the lemma is true up to a certain $k-1$, and following a similar reasoning as for the computation of $f_2(z)$, we have

$$f_k(z) = \int_z^1 f_1(x)\frac{f_{d-1}(z/x)}{x}dx = \int_z^1 2K_{k-1}\frac{z}{x}(\ln(z/x))^{k-2}dx \ .$$

By the change of variable $y = z/x$, we get

$$\begin{aligned} f_k(z) &= 2K_{k-1}z\int_z^1 \frac{(\ln y)^{k-2}}{y}dy = 2K_{k-1}z\left[\frac{(\ln y)^{k-1}}{(k-1)}\right]_z^1 dy \\ &= -\frac{2K_{k-1}}{(k-1)}z(\ln z)^{k-1} \ . \end{aligned}$$

□

In Figure 4 (bottom) we can see a plot of the functions $f_3(z) = 4z(\ln z)^2$ and $f_4(z) = -\frac{8}{3}z(\ln z)^3$. As expected, the higher k, the closer is the probability distribution to the left.

Let $f(z)$ be any continuous probability distribution for the asymptotic shape of the weights ω_i of a recurrence like $C_n = 1 + \sum_{i=0}^{n-1}\omega_i C_i$. As we have already seen, the solution is $C_n \sim c\ln n$ for some constant c, which can be computed by solving $c\ln n = 1 + \int_0^1 f(z)c\ln(zn)dz$. Once the terms $c\ln n$ cancel from both sides, what we have left is

$$c = -\frac{1}{\int_0^1 f(z)\ln z dz} \ . \tag{2}$$

The following technical lemma will be useful.

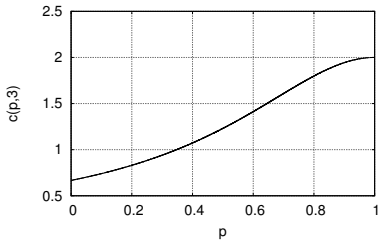

Figure 5: $c(p, 3)$

Lemma 2. *For $k \geq 0$, let $I_k = \int_0^1 z(\ln z)^k dz$. Then $I_k = -k!/(-2)^{k+1}$.*

PROOF. By induction. For the base case, we have $I_0 = 1/2$. For $k \geq 1$,

$$I_k = \int_0^1 z(\ln z)^k dz = \left[\frac{z^2(\ln z)^k}{2}\right]_0^1 - \int_0^1 \frac{k}{2} z(\ln z)^{k-1} dz = -\frac{k}{2} I_{k-1} \ .$$

□

By combining this lemma with Equation 2, we can easily compute the constant of searching in a quad-tree with k dimensions:

$$c(k) = -\frac{1}{\int_0^1 K_k z(\ln z)^k dz} = -\frac{1}{(K_k I_k)} = \frac{2}{k} \ ,$$

as expected. Therefore, the next theorem follows.

Theorem 2. *The expected* IPL *of a random quad tree of size n is $\sim \frac{2}{k} n \ln n$.*

Now we are ready for the analysis of some of the new variants presented in this paper. For instance, consider a 3-dimensional quad-kd tree built using RS. Let p be the probability of zero, and $q = 1 - p$ be the probability of one. Then, with probability q^3 the current node discriminates w.r.t. the three dimensions, with probability $3pq^2$ w.r.t. two dimensions, and with probability $3p^2q + p^3$ w.r.t. one dimension. Therefore, $f(z) = q^3 f_3(z) + 3pq^2 f_2(z) + (3p^2q + p^3) f_1(z)$, and the constant of the search is thus

$$c(p, 3) = \frac{-1}{q^3 k_3 I_3 + 3pq^2 k_2 I_2 + (3p^2q + p^3) k_1 I_1} = \frac{2}{p^3 - 3p + 3} \ .$$

We can see $c(p, 3)$ in Figure 5.

Generalizing the previous reasoning to the case of k dimensional quad-kd trees, it is easy to obtain that for a k-dimensional quad-kd tree built using RS,

$$c(p, k) = \frac{2}{k(1-p) + p^k} \ . \tag{3}$$

Therefore, the next theorem follows.

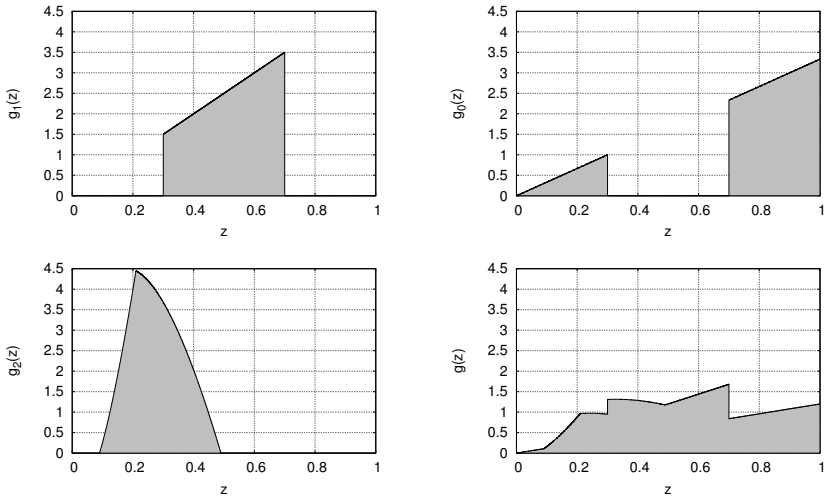

Figure 6: $g_1(z)$ (top left), $g_0(z)$ (top right), $g_2(z)$ (bottom left) and $g(z)$ (bottom right) for $t = 0.3$ and $k = 2$

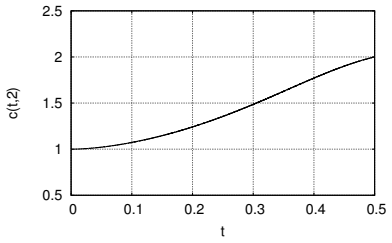

Figure 7: $c(t, 2)$

Theorem 3. *The expected* IPL *of a random quad-kd tree of size n built under the* RS *insertion heuristic is $\sim c(p,k)n \ln n$, where $c(p,k) = \frac{2}{k(1-p)+p^k}$.*

Let us turn to the analysis of the *IPL* of *DD* under random keys. Note that replacing p by $2t$ (where t is the split threshold) is not enough[3]. For instance, assume $t = 0.3$. It is true that the probability of discriminating is $1 - 2t = 0.4$. However, given that we discriminate by one dimension, the shape function for the weights is not $f_1(z)$ as in Figure 4.

Let $u = 1 - t$ and $a = 2/(u^2 - t^2) = 2/(1 - 2t)$. The shape function is $g_1(z) = az$ for $t \le z \le u$, with $g_1(z) = 0$ otherwise. (See Figure 6, top left.) For example, if $D = [0, 1]^k$ and we discriminate by a coordinate 0.45, then $\sim 0.45n$

[3]This mistake was made in a previous version of this paper [6].

keys will be inserted in the left subtree, and $\sim 0.55n$ keys will be inserted in the right subtree. Moreover, a is such that $\int_0^1 g_1(z)dz = 1$.

Using a similar reasoning, the shape function for the weights when we do not discriminate by any dimension is $g_0 = z/t$ for $z \leq t$ and for $z \geq u$, and $g_0(z) = 0$ otherwise. (See Figure 6, top right.)

At this point we can already compute the constant of the *IPL* for *DD* when $k = 1$. We have $g(z) = 2tg_0(z) + (1-2t)g_1(z) = 2z$. Therefore, $c = -1/\int_0^1 2z \ln z dz = 2$ for every t, as expected.

Calculating the constant for $k = 2$ is more intricate. We need to compute $g_2(z)$, the shape function when both coordinates are used to discriminate. Let $g_{2a}(z) = 4z\ln(z/t^2)/(1-2t)^2$ and $g_{2b}(z) = 4z\ln(u^2/z)/(1-2t)^2$. To begin with, we have $g_2(z) = 0$ for $z < t^2$ and for $z > u^2$. For $t^2 \leq z \leq tu$, we have

$$g_2(z) = \int_t^{z/t} g_1(x)\frac{g_1(z/x)}{x}dx = a^2 z \int_t^{z/t} \frac{1}{x}dx = g_{2a}(z).$$

Finally, for $tu \leq z \leq u^2$, we have

$$g_2(z) = \int_{z/u}^{u} g_1(x)g_1(z/x)/x dx = g_{2b}(z).$$

The function $g_2(z)$ for $t = 0.3$ can be seen at the bottom left of Figure 6.

Altogether, the shape function for $k = 2$ (see Figure 6, bottom right) is

$$g(z) = 4t^2 g_0(z) + 4t(1-2t)g_1(z) + (1-2t)^2 g_2(z).$$

Now we can compute $I = \int_0^1 g(z)\ln z dz$. The function $g_0(z)$ contributes to I with the term $4t^2\left(\int_0^t g_0(z)\ln z dz + \int_u^1 g_0(z)\ln z dz\right)$, $g_1(z)$ contributes with $4t(1-2t)\int_u^t g_1(z)\ln z dz$, and finally function $g_2(z)$ contributes with the terms $(1-2t)^2\left(\int_{t^2}^{tu} g_{2a}(z)\ln z dz + \int_{tu}^{u^2} g_{2b}(z)\ln z dz\right)$. After some calculations and simplifications, we can conclude that

$$c(t,2) = \frac{-1}{I} = \frac{1}{1 - 2tu + 2t^2 u\ln t - 2u^3 \ln u} \ .$$

Figure 7 shows $c(t,2)$. Altogether, the next theorem follows.

Theorem 4. *The expected* IPL *of a random quad-kd tree of size n built under the* DD *insertion heuristic is $\sim c(t,k)n\ln n$. For the case $k = 2$, we have $c(t,2) = \frac{1}{1-2tu+2t^2u\ln t - 2u^3\ln u}$, where $u = 1-t$.*

To check the analytical results above, looked at our experiments using *RS* and *DD* to measure the *IPL* and the number of empty subtrees as a function of n. The experimental results could (wrongly) suggest that both metrics are linearly proportional to n. This is true for the latter (see next section), but false for the former.

p	$c(p,2)$	$c'(p,2)$	$c(p,3)$	$c'(p,3)$	$c(p,4)$	$c'(p,4)$	$c(p,5)$	$c'(p,5)$
0.0	1.00	-0.62	0.66	-0.00	0.50	0.19	0.39	0.38
0.1	1.09	-0.72	0.73	-0.05	0.55	0.13	0.43	0.31
0.2	1.21	-0.91	0.83	-0.28	0.63	-0.02	0.49	0.22
0.3	1.34	-1.28	0.94	-0.51	0.70	-0.02	0.56	0.20
0.4	1.45	-1.41	1.07	-0.68	0.82	-0.24	0.65	0.09
0.5	1.60	-1.88	1.22	-0.96	0.97	-0.50	0.76	0.03
0.6	1.71	-2.02	1.44	-1.65	1.17	-0.97	0.95	-0.39
0.7	1.83	-2.39	1.58	-1.61	1.37	-1.16	1.18	-0.76
0.8	1.93	-2.75	1.75	-1.92	1.64	-1.87	1.51	-1.62
0.9	1.96	-2.61	1.94	-2.71	1.88	-2.47	1.81	-2.21
1.0	2.00	-2.91	1.99	-2.78	2.00	-2.91	1.99	-2.82

Table 2: The coefficients $c(p,k)$ and $c'(p,k)$ of the *IPL* of a quad-kd tree built under *RS*

Note that the first term of the *IPL* is $c(p,k)n\ln n$, while the second term could have a similar growth, even if it were of the $O(n)$ kind. To check this by practical means, we assumed that the *IPL* is $\sim c(p,k)n\ln n + c'(p,k)n$, fixed two large values for n (call them N_1 and N_2), computed experimentally the *IPL* for both values of n (call them I_1 and I_2), and solved the system of two linear equations

$$\begin{aligned} c(p,k)N_1\ln N_1 + c'(p,k)N_1 &= I_1 \\ c(p,k)N_2\ln N_2 + c'(p,k)N_2 &= I_2 \end{aligned}$$

Table 2 includes the experimental approximations that we obtained using this method. Note that we do not have a formal proof that the second term of the *IPL* is $c'(p,k)n$. However, the perfect matching of our theoretical formula for $c(p,k)$ with the experimental values seems a strong argument for the existence of $c'(p,k)$, and for the correctness of the provided values (up to some precision).

On the other hand, the theoretical results above and our experiments also indicate that increasing the number of dimensions has considerable effect on the results. Indeed, for high dimensions, quad-kd trees can avoid the waste of space of quad trees while keeping a short *IPL*.

6. Number of empty subtrees

Let us start computing ℓ, the expected number of leaves (empty subtrees) of random quad-kd trees of size n built using *RS* and *DD*.

Theorem 5. *The expected number of empty subtrees of random quad-kd trees of size n built using* RS *and* DD *is respectively*

$$\ell = \ell(p,k) = \left((2-p)^k + p^k - 1\right)n + 1,$$

and

$$\ell = \ell(t,k) = \left((2-2t)^k + (2t)^k - 1\right)n + 1 \ .$$

PROOF. Since every node except the root and every empty subtree is pointed to by one pointer, the total number of pointers is $n + \ell - 1$.

Furthermore, $\binom{k}{i}(1-p)^i p^{(k-i)}$ is the probability that a node contains 2^i pointers for every $1 < i \leq k$, and $p^k + \binom{k}{1}(1-p)p^{(k-1)}$ is the probability that a node contains exactly 2 pointers. Altogether, the average number of pointers per node is

$$2p^k + \sum_{i=1}^{k} 2^i \binom{k}{i} p^{(k-i)}(1-p)^i = (2-p)^k + p^k \ .$$

Therefore, we can immediately deduce

$$\ell = \ell(p,k) = \left((2-p)^k + p^k - 1\right) n + 1 \ . \tag{4}$$

As for DD, exactly the same reasoning applies, replacing p by $2t$. Hence,

$$\ell = \ell(t,k) = \left((2-2t)^k + (2t)^k - 1\right) n + 1 \ . \tag{5}$$

□

From the results of this section and the previous one, it is easy to see that playing with the values of p for RS (and t for DD) we can produce the whole range of values for the IPL and the number of empty subtrees, from those of kd trees with $p = 1$, to those of quad trees with $p = 0$.

Moreover, if one wants to go further onto the saving of space, the classic technique of *bucketing* is very useful for quad-kd trees. Indeed, fixing a constant value b as bucket size and storing in just one node all the records of a quad-kd subtree of size at most b, the savings on space are considerable.

It is out of the scope of this paper to enter into the details on how to implement bucketing. However, for a better understanding, we should say that buckets make sense to appear at the leaves level since it is there where the wastes in terms of null pointers are present. Of course it is possible to have buckets all over the tree but far from the leaves the savings using them are not significant.

In general, the number of leaves ℓ of a quad-kd tree with n nodes is $O(n)$. Let us consider RS. We now assume that there is a constant $\varepsilon(p,k,b)$ such that $\ell = \varepsilon(p,k,b) \cdot n + o(n)$. Note that Equation 4 shows $\varepsilon(p,k,0)$.

Fix p and k, and imagine that the booleans of the split vector of the root are decided from right to left. Let $\varphi(n,j)$ be the expected number of leaves of a quad-kd tree of size n when the random choices so far have set the variables $s_j = \ldots = s_{k-1} = 0$. We have the following recurrence.

$$\varphi(n,j) = \begin{cases} 1 & n \leq b \\ \frac{2}{n} \sum_{m=0}^{n-1} \varphi(m,k) & n > b, j = 0 \\ p\varphi(n,j-1) + (1-p)\frac{2}{n} \sum_{m=0}^{n-1} \varrho(m,j-1) & \text{otherwise} \end{cases}$$

p	$\frac{\varepsilon(p,2,5)}{\varepsilon(p,2,0)}$	$\frac{\varepsilon(p,3,5)}{\varepsilon(p,3,0)}$	$\frac{\varepsilon(p,4,5)}{\varepsilon(p,4,0)}$	$\frac{\varepsilon(p,5,5)}{\varepsilon(p,5,0)}$
0	0.17	0.12	0.09	0.07
0.1	0.18	0.13	0.10	0.08
0.2	0.19	0.14	0.11	0.09
0.3	0.21	0.16	0.12	0.10
0.4	0.22	0.17	0.14	0.12
0.5	0.24	0.19	0.16	0.13
0.6	0.25	0.22	0.18	0.16
0.7	0.26	0.24	0.21	0.19
0.8	0.27	0.26	0.24	0.23
0.9	0.28	0.28	0.27	0.26
1.0	0.28	0.28	0.28	0.28

Table 3: Proportion of the pointers used by a quad-kd tree built under RS with buckets of size 5 with respect to the pointers used without bucketing

where $\varrho(n,j)$ is the expected number of leaves of a quad-kd tree of size n when the random choices so far have set $s_j + \ldots + s_{k-1} > 0$. The recurrence for $\varrho(n,j)$ is

$$\varrho(n,j) = \begin{cases} 1 & n \le b, j = 0 \\ p\varrho(0,j-1) + 2(1-p)\varrho(0,j-1) & n \le b, j > 0 \\ \varphi(n,k) & n > b, j = 0 \\ p\varrho(n,j-1) + (1-p)\frac{2}{n+1}\sum_{m=0}^{n}\varrho(m,j-1) & \text{otherwise} \end{cases}$$

Solving all these recurrences is beyond the purpose of this paper. However, a dynamic-programming implementation allows us to compute the exact values of $\varphi(n,k)$ and $\varrho(n,k)$ up to a certain value of n for several values of k. The results suggest that $\varepsilon(p,k,b) \sim \varphi(n,k)/n$ exists and converges quickly. In Table 3 we can see some ratios to emphasize the huge savings that can be achieved by this simple technique.

Let us compute the savings of bucketing with DD, following similar steps. Fix t and $u = 1 - t$. Without loss of generality, assume $D = [0,1]^k$. Let $I_a^b(z) = \binom{a+b}{a}\int_0^z x^a(1-x)^b dx$. We will need to evaluate $I_a^b(t)$, $I_a^b(u)$ and $I_a^b(1)$ for combinations of a and b such that $0 \le a + b \le n$. There does not seem to be a simple closed expression for those integrals. However, for the sake of computation, we can use the recurrence

$$I_a^b(z) = \frac{(a+b)}{b}\cdot I_a^{b-1}(z) - \frac{(a+1)}{b}\cdot I_{a+1}^{b-1}(z)$$

for $b > 0$, with $I_z(a,0) = z^{a+1}/(a+1)$.

Let $J_m^n(z) = I_m^{n-m-1}(z)$. Assume that we discriminate by the current coordinate j. This happens with probability $u - t$. The value x_j for this coordinate at the current root can be anyone between t and u, with a constant density of

t	$\frac{\varepsilon(t,2,5)}{\varepsilon(t,2,0)}$	$\frac{\varepsilon(t,3,5)}{\varepsilon(t,3,0)}$	$\frac{\varepsilon(t,4,5)}{\varepsilon(t,4,0)}$
0	0.17	0.12	0.09
0.1	0.17	0.16	0.10
0.2	0.2	0.15	0.12
0.3	0.22	0.18	0.15
0.4	0.25	0.23	0.21
0.5	0.28	0.28	0.28

Table 4: Proportion of the pointers used by a quad-kd tree built under *DD* with buckets of size 5 with respect to the pointers used without bucketing

probability $1/(u-t)$. Therefore, the probability that exactly m keys will be inserted into the left child is

$$\int_t^u \binom{n-1}{m} \frac{x^m(1-x)^{n-m-1}}{u-t} dx = \frac{J_m^n(u) - J_m^n(t)}{u-t} \quad .$$

Assume now that we have not discriminated by any of the k coordinates, which happens when $x_j \notin [t,u]$ for every j. When we choose a random j to discriminate, x_j can be anyone between 0 and t or between u and 1, with a constant density of probability $1/(2t)$. Therefore, the probability that exactly m keys will be inserted into the left child is

$$\frac{J_m^n(1) - J_m^n(u) + J_m^n(t)}{2t} \quad .$$

Altogether, we have the following recurrence for *DD*.

$$\varphi(n,j) = \begin{cases} 1 & n \leq b \\ \frac{1}{t} \sum_{m=0}^{n-1} \left(J_m^n(1) - J_m^n(u) + J_m^n(t)\right) \varphi(m,k) & n > b, j = 0 \\ 2t\varphi(n,j-1) + 2 \sum_{m=0}^{n-1} \left(J_m^n(u) - J_m^n(t)\right) \varrho(m,j-1) & \text{otherwise} \end{cases}$$

By a similar reasoning, the recurrence for $\varrho(n,j)$ is

$$\varrho(n,j) = \begin{cases} 1 & n \leq b, j = 0 \\ 2t\varrho(0,j-1) + 2(u-t)\varrho(0,j-1) & n \leq b, j > 0 \\ \varphi(n,k) & n > b, j = 0 \\ 2t\varrho(n,j-1) + 2 \sum_{m=0}^{n} (J_m^{n+1}(u) - J_m^{n+1}(t))\varrho(m,j-1) & \text{otherwise} \end{cases}$$

The results suggest that $\varepsilon(t,k,b) \sim \varphi(n,k)/n$ exists and converges to the values presented in Table 4.

As shown in Tables 3 and 4, and as expected, for both *RS* and *DD*, the savings using buckets grow with k, as well as with the order of the split vector, so the savings are more important when the quad-kd trees more closely resemble

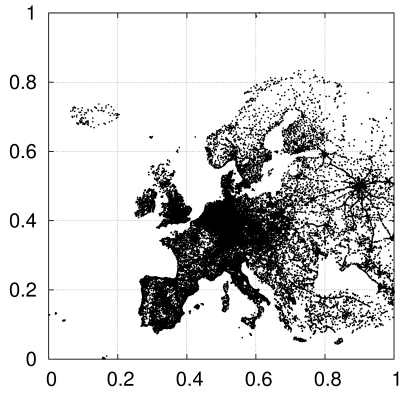

Figure 8: Fuel stations in Europe

quad trees. It can also be observed that the savings are already important with a relatively small value of b. Note that in a practical setting, the optimal value for b should be limited by $\Theta(\log n)$ but also by the size of a cache block.

One final comment is in order. The field `elems` for the nodes of the code in Program 1 was included to be able to use buckets. A shown in the program, updating it is trivial. It could be argued that this field is a waste of space on its own, but in a 64-bit machine the size of an integer is usually half the size of a pointer, so the extra memory is by far less than the memory used without bucketing. Moreover, this extra information can be used to perform (and/or improve the efficiency) of several operations on quad-kd trees, such as rank operations.

7. Experiments with real data

To study the performance of quad-kd trees with real input data, we used the 2-dimensional coordinates of the 87200 fuel stations of Europe –taken from *OpenStreetMap* (`http://www.openstreetmap.org/`) and projected to the $[0, 1]^2$ domain. The resulting plot can be appreciated in Figure 8.

The corresponding experimental results were obtained building quad-kd trees using *RS* and *DD* and averaging over ten repetitions, with these input points in different random order. Figure 9 shows the competitive average performance of both heuristics.

The difference between these experimental results and the ones for uniformly and independently generated points is that the latitude and longitude coordinates of real data points are not independent of each other, while for our theoretical analysis the independence requirement is mandatory. However, it is worth noting that the *IPL* and the number of empty subtrees for real data is

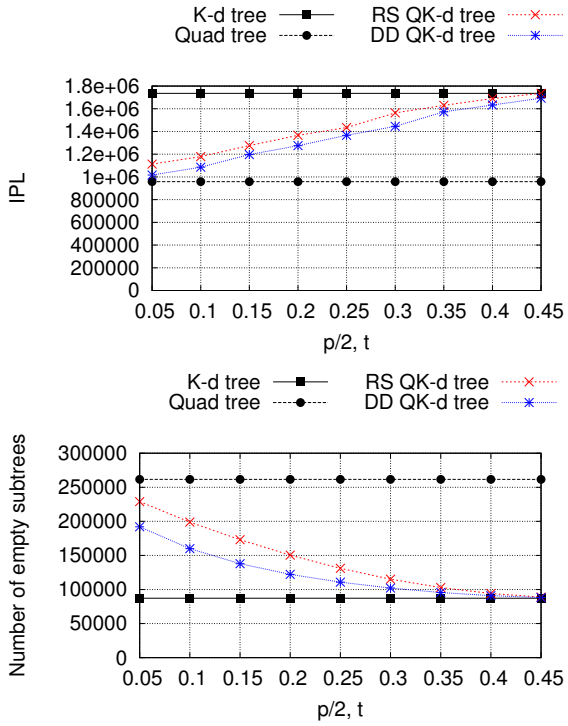

Figure 9: Performance of *RS* and *DD* with real input data

very similar to those of ideally constructed points (shown in Figure 2), except that here *DD* clearly outperforms *RS*.

8. Conclusions

We have introduced a simple and flexible multidimensional data structure, the quad-kd tree, which includes kd trees and quad trees as particular cases. We have shown through formal analysis and experiments that, as expected, the performance of random quad-kd trees is between the one of random quad trees and the one of random kd trees considering *IPL* and number of empty subtrees.

We have also proposed three insertion heuristics that allow "a la carte" space and time trade-offs. Among these heuristics, *Random-Split* is the simplest, yet it is adjustable and has a stable performance in all the examined cases. On the other hand, the results for the *Distance-Dependent* heuristic are more favorable in the case of uniform data distributions. Finally, if the distribution of the input data is neither uniform nor independent, but still known, then the *Probability-Dependent* heuristic is preferable. Using these heuristics, the resources used by the data structure can be tailored to the application requirements. We

show, additionally, that using the bucketing technique it is possible to save a considerable amount of space.

As open work, a first line of research regarding the *Distance-Dependent* heuristic is whether it should be possible to compute $c(t, k)$ for $k \geq 3$ following steps similar to the ones presented for $c(t, 2)$.

A second—and challenging—line of future work consists of a formal analysis approach that takes quad-kd trees as a framework to analyze fundamental properties (average cost of update and search operations, and so on) of the whole family of hierarchical multidimensional trees akin to kd trees and quad trees.

A third line of further interest addresses the possible practical applicability of quad-kd trees. It may include the design of competitive heuristics for special input data sets, the design of self-adjustable heuristics for long sequences of biased associative queries, the exploration of alternative representations of quad-kd trees (e.g. succinct) and, the study of the performance of quad-kd trees under other memory models (e.g. cache).

References

[1] C.-H. Ang and H. Samet. Node Distribution in a PR Quadtree. In *Proceedings of 1st International Symposium on Large Spatial Databases*, pages 233–252, 1989.

[2] Benoit, Demaine, Munro, Raman, Raman, and Rao. Representing trees of higher degree. *Algorithmica*, 43, 2005.

[3] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] J. L. Bentley and J. H. Friedman. Data Structures for Range Searching. *ACM Computing surveys*, 11(4):397–409, 1979.

[5] J.L. Bentley and R. A. Finkel. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[6] Nikolett Bereczky, Amalia Duch, Krisztián Németh, and Salvador Roura. Quad-K-d Trees. In Alberto Pardo and Alfredo Viola, editors, *LATIN*, volume 8392 of *Lecture Notes in Computer Science*, pages 743–754. Springer, 2014.

[7] P. Chanzy, L. Devroye, and C. Zamora-Cura. Analysis of Range Search for Random k-d Trees. *Acta Informatica*, 37:355–383, 2001.

[8] H.-H. Chern and H.-K. Hwang. Partial Match Queries in Random k-d Trees. *SIAM Journal on Computing*, 35(6):1440–1466, 2006.

[9] Myung Geol Choi, Eunjung Ju, Jung-Woo Chang, Jehee Lee, and Young J. Kim. Linkless Octree Using Multi-Level Perfect Hashing. *Comput. Graph. Forum*, 28(7):1773–1780, 2009.

[10] W. Cunto, G. Lau, and Ph. Flajolet. Analysis of KDT-Trees: KD-Trees Improved by Local Reorganisations. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Workshop on Algorithms and Data Structures (WADS'89)*, volume 382 of *Lecture Notes in Computer Science*, pages 24–38. Springer-Verlag, 1989.

[11] Darragh, Cleary, and Witten. Bonsai: A Compact Representation of Trees. *SOFTPREX: Software–Practice and Experience*, 23, 1993.

[12] L. Devroye, J. Jabbour, and C. Zamora-Cura. Squarish k-d Trees. *SIAM Journal on Computing*, 30:1678–1700, 2000.

[13] A. Duch, V. Estivill-Castro, and C. Martínez. Randomized K-dimensional Binary Search Trees. In K. Y. Chwa and O. H. Ibarra, editors, *9th Annual International Symposium on Algorithms and Computation (ISAAC 98)*, volume 1533 of *Lecture Notes in Computer Science*, pages 199–208, 1998.

[14] A. Duch and C. Martínez. On the Average Performance of Orthogonal Range Search in Multidimensional Data Structures. *Journal of Algorithms*, 44(1):226–245, 2002.

[15] A. Duch and C. Martínez. Improving the Performance of Multidimensional Search Using Fingers. *ACM Journal of Experimental Algorithms - JEA*, 10, 2005.

[16] A. Duch and C. Martínez. Updating Relaxed K-d Trees. *ACM Transactions on Algorithms*, 6(1), 2009.

[17] C. Duncan, M.T. Goodrich, and S. Kobourov. Balanced Aspect Ratio Trees: Combining the Advantages of k-d Trees and Octrees. *Journal of Algorithms*, 38:303–333, 2001.

[18] D. Eppstein, M.T. Goodrich, and J. Z. Sun. Skip Quadtrees: Dynamic Data Structures for Multidimensional Point Sets. *Int. J. Comput. Geom. Appl.*, 18, 2008.

[19] Ph. Flajolet and C. Puech. Partial Match Retrieval of Multidimensional Data. *Journal of the ACM*, 33(2):371–407, 1986.

[20] Flajolet, Ph. and Gonnet, G. H. and Puech, C. and Robson, J. M. Analytic Variations on Quad-Trees. *Algorithmica*, 10:473–500, 1993.

[21] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[22] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving Hashing in Multidimensional Spaces. In *STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625, 1997.

[23] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison–Wesley, 2nd edition, 1998.

[24] H. M. Mahmoud. *Evolution of Random Search Trees*. Wiley-Interscience series in discrete mathematics and optimization, 1991.

[25] H. M. Mahmoud and B. Pittel. Analysis of the Space of Search Trees Under the Random Insertion Algorithm. *Journal of Algorithms*, 10:52–75, 1989.

[26] C. Martínez, A. Panholzer, and H. Prodinger. Partial Match Queries in Relaxed Multidimensional Search Trees. *Algorithmica*, 29(1–2):181–204, 2001.

[27] D. P. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC. Computer & Information Science Series, 2005.

[28] K. Mulmuley. Randomized Multidimensional Search Trees: Lazy Balancing and Dynamic Shuffling. In *FOCS: 32nd Annual Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[29] M. H. Overmars and J. van Leeuwen. Dynamic Multi-dimensional Data Structures Based on Quad and K-d Trees. *Acta Informatica*, 17(3):267–285, 1982.

[30] E. Park and D. M. Mount. A Self-Adjusting Data Structure for Multidimensional Point Sets. *LNCS*, 7501:778–789, 2012.

[31] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 233–242. ACM Press, 2002.

[32] D. Rotem. Clustered Multiattribute Hash Files. In *PODS '89 Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 225–234, 1989.

[33] Roura, S. Improved Master Theorems for Divide-and-Conquer Recurrences. *J. ACM*, 48(2):170–205, 2001.

[34] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[35] R. Sedgewick and Ph. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison–Wesley, 1996.

[36] M. Sherk. Self-Adjusting k-ary Search Trees. *Journal of Algorithms*, 19(1):25–44, 1995.

[37] Markku Tamminen. The Extendible Cell Method for Closest Point Problems. *BIT Numerical Mathematics*, 22:27–41, 1982.

[38] V. K. Vaishnavi. Multidimensional Height-Balanced Trees. *IEEE Transactions on Computers*, 33(4):334–343, 1984.